US-PAT-NO: **6601023**

DOCUMENT-IDENTIFIER: US 6601023 B1

TITLE: Method for impact analysis of a model

DATE-ISSUED: July 29, 2003

INVENTOR-INFORMATION:

NAME COUNTRY	CITY	STATE	ZIP CODE	
Deffler; Tad A.	Boonton	NJ	N/A	N/A
Russo; Mark	Belle Mead	NJ	N/A	N/A
Beerbower; Thomas R.	Holland	PA	N/A	N/A

US-CL-CURRENT: 703/13, 703/22 , 707/6

ABSTRACT:

A method for determining the net change to a model as a result of a transaction. The method includes receiving a request for a net change of a model as a result of a transaction. The method also includes identifying a plurality of model change records associated with the transaction. The method also includes determining and displaying net change of the model based on each of the plurality of model change records associated with the transaction.

8 Claims, 12 Drawing figures

Exemplary Claim Number: 1

Number of Drawing Sheets: 11

---------- KWIC ---------

US Patent No. - PN (1):
6601023

Drawing Description Text - DRTX (4):
FIG. 3 illustrates a data model of an exemplary embodiment of a **meta model** of the present invention.

Detailed Description Text - DETX (4):
An embodiment of the present invention is implemented, for example, as a software module written in the C++ programming language which may be executed on a computer system such as computer system 101 in a conventional manner. Using well known techniques, the application software may be stored on data storage medium 109 and subsequently loaded into and executed within the computer system 101. Once initiated, the software of the preferred embodiment operates, for example, in the manner described below. Universal Modeling Architecture (UMA) is a data-driven modeling engine that could work in various problem domains based on an external definition of a **meta model** that may be provided by a developer and be extended to provide an UMA-based product. An external definition of a **meta model** is, for example, a series of descriptions of the types of objects that are to be found in the problem domain, and the

properties associated with each of these objects. These descriptions may be provided by invoking a set of functions exposed by the implementation, and passing in, via function parameters, the descriptive information. Exemplary problem domains may include: data modeling such as database tables, columns and indices; process modeling such as activities and arrows; access modeling such as data manipulation language statements and files; and component modeling such as interfaces, implementations, and dependencies.

Detailed Description Text - DETX (5):
In an exemplary embodiment of the present invention, the UMA-based product is a modeling tool. The UMA-based product may be an UMA-based application such as a user interface that includes UMA. The UMA-based product may also include an instance of an object/property model based on an external definition of the **meta model** provided by, for example, a developer.

Detailed Description Text - DETX (6):
In an exemplary embodiment of the present invention, as shown in FIG. 2, UMA 200 includes a **meta model** 210, object and property factory registries 260, object/property interface 280, transaction manager 285, log file 295, and object/property model 290. The meta. model 210 may include a semantic registry 220 including a plurality of sets of semantics 230 and a meta data manager 240 including object/property descriptions 250. The object/property descriptions 250 are sets of information describing the characteristics of an object or a property. In the case of objects, this may include its name, a human-readable piece of descriptive text, generalization information, information about what other types of objects may be contained within it. Generalization information, for example, describes refinement/subtyping such as synonyms information.

Detailed Description Text - DETX (7):
Object/property descriptions may be provided by the developer or user as the externally defined **meta model**. The **meta model** 210 is a description of the objects and properties of the problem domain to be solved and a plurality of sets of semantics 230 to be respectively invoked to change objects and properties when changes to such objects and properties are requested.

Detailed Description Text - DETX (8):
FIG. 3 shows an exemplary embodiment of **meta model** 210 of the present invention. As shown in FIG. 3, the **meta model** 210 includes a plurality of classes such as objects, properties and semantics and establishes relationships between objects, properties and semantics. Type 310 is associated with a type code key, name attribute and definition attribute. Type 310 can be, for example, a category 330 or an item 320. Category 330 is associated with a type code key and includes a plurality of items. **Category** membership 340 is associated with a **category** key and **member** key. Item 320 is associated with a type code key and can be a property or object.

Detailed Description Text - DETX (11):
As shown in FIG. 2, the meta data manager 240 of the **meta model** 210 receives meta data (e.g., description of objects and properties), for example, from a developer through an UMA-based application 225. The meta data is what are the objects and properties provided by a developer to solve the problem domain of the developer. The meta data manager 240 receives validation creation requests from object and property factory registries 260.

Detailed Description Text - DETX (12):
As shown in FIG. 2, the semantic registry 220 of the **meta model** 210 includes a plurality of sets of semantics 230 which may include, for example, a

predefined set and additional sets provided by the developer through an UMA-based application. Semantics 230 are modeling rules encapsulated in semantic objects that expose an interface, such as a pure virtual class, that hide the modeling engine from details of the semantics 230. The predefined semantic set provides rules to enforce the integrity of the underlying modeling engine and in an exemplary embodiment of the present invention, the predefined semantic set cannot be changed by a developer. The additional sets of semantics provided by the developer can be modified by the developer. An example of a semantic may be, for example, one that enforces the rule "No two columns in a single database table may have the same name."

Detailed Description Text - DETX (15):
As shown in FIG. 2, object/property interface 280 receives requests for the modification and deletion of objects and properties, for example, by a developer through UMA-based application 225. Such requests invoke the respective semantics in **meta model** 210 which may result in changes to the objects and properties which are provided to object/property interface 280 from semantic registry 220 of **meta model** 210. The object/property interface 280 may identify a discrete event from a plurality of discrete events, for example, as shown in Table One, and determine that a semantic or set of semantics should be invoked. The discrete events indicate occurrences where object/property model 290 may be modified. In an exemplary embodiment of the present invention, authors of UMA may provide a plurality of discrete events.

Detailed Description Text - DETX (17):
FIG. 4, illustrates a flow diagram of an exemplary method of object/property interface 280 firing an object semantic. In 410, the object/property interface 280 opens transaction with transaction manager 285. A transaction is opened for each change to the model so that the change can be cancelled if found to be invalid. In 415, object/property interface 280 communicates with the object itself. All objects know their type to determine the object type. In 420, object/property interface 280 obtains the complete supertype path for a respective object from **meta model** 210, for example, in top-down order. In 425, the first object type (ultimate supertype) is obtained by object/property interface 280 from the metamodel. In 430, object/property interface 280 obtains the semantic list for the object type from object semantic registry 220. In 435, object/property interface 280 communicates with semantic registry 220 to determine whether more semantics are in the list. If yes, object/property interface 280 obtains, in 455, and fires, in 460, the next semantic. The object/property interface 280 determines, in 465, if the semantic failed. If so, in 470, the transaction is rolled back by the transaction manager 285 pursuant to a request from object/property interface 280. If, however, the semantic did not fail, in 435, object/property interface 280 will again determine whether any more semantics are in the list.

Detailed Description Text - DETX (19):
FIG. 5 illustrates a flow diagram of an exemplary method of object/property interface 280 firing a property semantic. In 510, object/property interface 280 opens the transaction with transaction manager 285. In 515, object/property interface 280 communicates with the object to determine the object type. In 520, object/property interface 280 obtains the complete supertype path for the respective object from **meta model** 210, for example, in top-down order. In 525, the first object type (ultimate supertype) is obtained by object/property interface 280 from the metamodel. In 527, object/property interface 280 constructs an object/property pair for object and property types. In 530, object/property interface 280 obtains the semantic list for the object/property pair type from property semantic registry 530.

Detailed Description Text - DETX (26):
In an exemplary embodiment of the present invention, UMA may work in an

access modeling problem domain such as data manipulation language statements (DML) and files. The object/property model 290, for example, may include as objects, Table A, Table B, DML statements such as SELECT A.x+B.y from A, B (where A is Table A, A.x refers to column x in Table A, B is Table B, and B.y refers to column y in Table B), and File C (a file that uses the SELECT A.x+B.y from A, B DML statement). These objects, for example, would be included in the object/property model 290. The semantics 230 of the **meta model** 210 would include, for example, semantics describing how changes to the object/property model 290 affects the respective DML statements. For example, semantic (1) may provide that the deletion of a table should cause all references to the table to be removed and surrounding syntax corrected from respective DML statements. Semantic (2) may provide that all objects in the object/property model 290 that represent files that use modified or deleted DML statements being designated as touched. A touched object or property is marked as logically modified even if no actual modification has occurred to it.

Detailed Description Text - DETX (31):
In 620, a **meta model** 210 is constructed, for example, by defining a second set of classes. The second set of classes are designed to hold descriptions of properties, objects and semantics. For example, in an exemplary embodiment of the present invention, two semantic interface classes are specified. A first semantic interface class, for example, UMEObjectSemanticI, is the interface for any semantic that affects the creation or destruction of an object. Further, a second semantic interface class, for example, UMEPropertySemanticI, is the interface for any semantic that affects the creation, destruction, or modification of a property. In an exemplary embodiment of the present invention, the implementation of a representation of **meta model** 210 includes a singleton object that exposes static methods for registering meta data and semantics.

Detailed Description Text - DETX (32):
Modeling rules, e.g., semantics 230, may be incorporated into semantic registry 220 of **meta model** 210 as a predefined set of semantics as in 640 and/or as additional sets of semantics, for example, provided by a developer either upon startup or any time thereafter. In order to incorporate a modeling rule into **meta model** 210 for an object, the developer subclasses the appropriate calls, for example, UMEObjectSemanticI for a modeling rule for an object and UMEPropertySemanticI for a modeling rule for a property. The developer also could implement a fire method to perform the desired operations and checks upon object/property model 290. A value will be returned to indicate whether the operation was successful. For example, a return value of TRUE would indicate that the operation was successful, and a return value of FALSE would indicate that the operation could not be performed successfully or that the model was in an invalid state. Access points (e.g., object/property model modification points) may also be included in **meta model** 210 for invoking semantics 230. The various model modification points (e.g., CreateObject) would invoke the respective semantic at the appropriate point. If an indication is received from the firing of semantics that one or more semantics had failed to complete successfully, for example, FALSE, the operation could then abort.

Detailed Description Text - DETX (33):
In 630, a type code is associated with the first and second set of classes. A type code is a unique identifier that specifies what type of meta data is being represented. Each item specified in the **meta model,** both objects and properties, would have a unique type code. In an exemplary embodiment of the present invention, UMA 200 includes a variable type, TypeCode_t, which is declared to hold the type codes. A predefined set of semantics is provided to the semantic registry 220 of **meta model** 210 in 640.

US-PAT-NO: 6601072

DOCUMENT-IDENTIFIER: US 6601072 B1

TITLE: Method and system for distribution of application data to distributed databases of dissimilar formats

DATE-ISSUED: July 29, 2003

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Gerken, lll; John Kenyon	Raleigh	NC	N/A	N/A

US-CL-CURRENT: 707/103R, 707/100 , 707/101 , 707/4

ABSTRACT:

A system and method for distribution of application program data to update a plurality of databases of dissimilar formats, including an application access program having a business object, a storage object and a data access object that is associated with a database. The business object creates data objects as needed and sends database access requests to the storage object. Each data access object registers with the storage object, and the storage object notifies each data access object when a data access request is received. The storage object then sends the data object to the data access objects. Data encapsulated in the data object is converted into a format for each database in which the data is to be stored. When a read request is received, data is read from a database that is designated as the primary database. When a write request is received, data is formatted into multiple formats and one format is written into each database.

18 Claims, 4 Drawing figures

Exemplary Claim Number: 1

Number of Drawing Sheets: 4

---------- KWIC ---------

Drawing Description Text - DRTX (4):
FIG. 2 is an exemplary Unified Modeling Language (UML) class diagram illustrating the **relationships between different objects** of the application data distribution system in accordance with a preferred embodiment of the present invention.

Detailed Description Text - DETX (4):
FIG. 2 is an exemplary Unified Modeling Language (UML) class diagram illustrating the **relationship between different objects** in a preferred embodiment of the present invention. UML has two major components--a **meta-model** and a notation. The **meta-model** describes the objects, attributes and relationships necessary to represent the concepts of UML within a software application. There is a notation for modeling the dynamic elements of a design such as objects, messages, and finite state machines.

Detailed Description Text - DETX (8):

An association relationship is used to depict forms of containment that do not have whole/part implications. A line drawn between two participating classes represents an association. If the association has an arrowhead, this indicates that the class at the arrowhead end does not know anything about the attached class. Sometimes the **relationship between two classes** is very weak. A dashed arrow represents a dependency **relationship between a client class** and a supplier class to show that the client class depends on the supplier class to provide certain services.

Detailed Description Text - DETX (10):
The business object 12 can have a plurality of methods defined and among these methods there are, for example, read( ), write( ) and createDataObject( ). The business object 12, through its createDataObject( ) method, can request that a data object 20 be created by the data object factory 22. The business object 12 can handle multiple instances of data object 20. It is a "one to many" (1:*) **relationship between the business object** 12 and data object 20. Data object 20 represents some logical grouping of information that is applicable to an application. For instance, a person data object for an address book application contains all relevant information about the person that the address book tracks. Data object 20 is created by data object factory 22 based on the needs of a business object 12.

Detailed Description Text - DETX (21):
Although not critical to the present invention, storage **objects** 24 are preferably created by factories and have virtual base **classes** that define their common **behavior**. Data access objects 34, 36 are also preferably created by factories and multiply inherit from two virtual base classes. The first base class defines behavior common to local DAOs 34 or remote DAOs 36. The second base class defines behavior common to the database type with which the DAO 34, 36 interface.

Detailed Description Text - DETX (22):
A data access object has primarily two defining characteristics--the database that it is responsible for manipulating and the types of data objects that it is programed to receive and handle. For a given application, there can be a plurality of DAOs each responsible for updating database "X", each with a different type of data derived from different data objects "Ys". Furthermore, there can be a plurality of DAOs responsible for handling a data object "Y" with each responsible for updating a different database "X". Therefore, it is appropriate to design the **object** model such that common **behavior** for database "X" is declared and defined in a base **class** for that database or database type. It is also appropriate to define a base **class** which describes common **behavior** for handling data **objects** of a certain type. This creates a two dimensional matrix for defining behavior and therefore describes a multiple inheritance design for the storage subsystem. A given DAO intended to handle certain data **objects** and update a certain database would inherit from the data **object class** and database base **class** appropriate for its task, and then determine the additional required **behavior** in the new subclass.

95 shts and printed yet

PGPUB-DOCUMENT-NUMBER: 20030202645

PGPUB-FILING-TYPE: new

DOCUMENT-IDENTIFIER: US 20030202645 A1

TITLE: Element management system with adaptive interface based on autodiscovery from element identifier

PUBLICATION-DATE: October 30, 2003

INVENTOR-INFORMATION:

NAME	CITY	STATE	COUNTRY
RULE-47			
Naik, Dharmendra	San Jose	CA	US
Levesque, Gilbert	Mountain View	CA	US
Galou, Salim	San Jose	CA	US
Wong, Malais	Sunnyvale	CA	US

US-CL-CURRENT: 379/201.1

ABSTRACT:

A network element management system which automatically configures itself, when an operator enters a component identifier, for optimal full-featured management of the identified component. This is preferably done by an autodiscovery process, and not by any mere lookup.

---------- KWIC ---------

Brief Description of Drawings Paragraph - DRTX (42):
[0074] FIG. 38 shows a **meta model** class diagram.

Detail Description Paragraph - DETX (1159):
[1236] Introduction to **Meta Model**

Detail Description Paragraph - DETX (1160):
[1237] This section provides overview and design information about **Meta Model** component of NETSMART. The **Meta Model** provides an instantiated means of describing any specific network element and generic. The **meta model** captures the physical and logical components that comprise the NE.

Detail Description Paragraph - DETX (1161):
[1238] This **meta model** is used by the application to access meta information for an NE and any specific instance of a component supported by the NE. One challenge and objective for NETSMART is to provide a generic solution for introducing new NE assemblies. The design of the **meta model** was intended to satisfy this goal. Other applications within NETSMART also face this similar challenge.

Detail Description Paragraph - DETX (1162):
[1239] It is been widely accepted that a network element suites a generic

assembly tree pattern very well. Ideally, applications like auto discovery, auditing, reconciliation, database change handling, configuration management should all provide generic solutions (i.e. non-type specific). Type specific traversal is supported by the **meta model** for those applications that find it difficult to design generically.

Detail Description Paragraph - DETX (1164):
[1241] This document describes the NE **Meta Model** design with some coverage on related applications (e.g. Autodiscovery). It does not cover details on aspects related to other components. It is acknowledged that additional requirements for the **Meta Model** will arise during the remaining development phases of NETSMART. The NE **Meta Model** is very flexible and extensible. It is expected that as additional requirements are identified they will be added to subsequent revisions of this document.

Detail Description Paragraph - DETX (1166):
[1243] This section provides an overview for the **Meta Model** component of NETSMART. **Meta Model** is the core the data driven approach of NETSMART. Various semantic details of a network element are captured in **Meta Model,** such as different objects managed by network element (e.g shelves, equipment, facilities etc.), **relationships between these objects,** various attribute applicable to these managed objects etc. The containment hierarchies between managed objects is also captured in **Meta Model**. The **meta model** provides a type of management information base for the application. If something changes between two successive releases of a network element, it can be handled in **Meta Model** by just incorporating these changes to the network element model. Following are few examples of the kind of information captured in this component:

Detail Description Paragraph - DETX (1171):
[1248] The following depict the major design objectives for the **meta model**.

Detail Description Paragraph - DETX (1173):
[1250] Allow the applications to traverse the **meta model** hierarchically and generically.

Detail Description Paragraph - DETX (1174):
[1251] Allow the applications to define various data model, and **meta model,** relationships which will help satisfy their specific requirements. Note: These may and probably will be unrelated to the assembly containment. For example, a MetaSection is a way to group MetaSlot(s) for alarm correlation but a MetaSection has nothing to do with the assembly hierarchy.

Detail Description Paragraph - DETX (1176):
[1253] Each managed object on network element is represented by a Meta Object in the **Meta Model** which contains all relevant meta information about that managed object. There are different Meta Objects for different type of facilities, equipment, EPGs etc. Each Meta Object for a given NE type in the **meta model,** is identified by Meta Instance Id which is just a label given to Meta Objects for identification. If there are multiple Meta Objects of the same type (e.g. two STS1 facilities) which have same characteristics, they are mapped to the same Meta Component. Meta Component represents `Type` of Meta Object. The attributes can be defined against a Meta Object as well as Meta Component. Attributes defined against a Meta Component are applicable to all Meta Objects related to that Meta Component. All common attributes of similar Meta Objects are defined against Meta Component whereas, if there are any specific attributes applicable to a particular Meta Object only, they can be

defined against Meta Object itself. **Meta Model** also captures some information which does not pertain to network element directly, for example the name of the java class to be used for address translation for a particular NE is also captured in **meta model**.

Detail Description Paragraph - DETX (1177):
[1254] In the overall picture of NETSMART, **meta model** resides in Application Server. Configuration Manager is the main user of **meta model**. All other components access **meta model** via Configuration Manager. The usage of **meta model** can be divided into two major parts:

Detail Description Paragraph - DETX (1180):
[1257] The **meta model** knows nothing about the current state of a network element. It maintains only a static representation of a network element. The Autodiscovery.sup.1 application uses the **meta model** to drive the building of the dynamic view of the network element. Only the dynamic view of a network element provides current state. The dynamic view of a network element is captured in another component namely the Configuration Manager. Fundamentally, the dynamic view of a network element is a subset of the **meta model** containment tree. During autodiscovery, the containment hierarchy modeled in **meta model** is used to build the containment tree (dynamic view) of application objects in Configuration Manager. The containment hierarchy in **meta model** dictates the containment hierarchy in configuration manager. Only containment relationships in **meta model** are used for autodiscovery. Other relationships in the **meta model** relationship tree are not used during autodiscovery. .sup.1 Autodiscovery is the process of obtaining configuration information from NE and persisting in Configuration Manager.

Detail Description Paragraph - DETX (1181):
[1258] Once autodiscovery is done, configuration manager can obtain any specific meta information from **meta model**. For example, while displaying equipment and ports in configuration manager GUI, configuration manager needs to find out various ports supported by a particular equipment. This information is obtained by configuration manager from **meta model** on need basis. Another example is that when user wants to change some provisioning attribute on a facility or equipment, configuration manager GUI presents list of valid values for that attribute. This list of valid values is obtained from **meta model** on need basis.

Detail Description Paragraph - DETX (1182):
[1259] **Meta Model** can be divided into two parts on broad basis:

Detail Description Paragraph - DETX (1183):
[1260] **Meta Model** Data

Detail Description Paragraph - DETX (1184):
[1261] **Meta Model** Engine

Detail Description Paragraph - DETX (1185):
[1262] **Meta Model** data contains all the static modeling information about the network element. It models all the Meta **Objects, relationships between** them, various applicable attributes on Meta Objects, valid values for these attributes etc. This data is captured in oracle database tables as per the data model. Initially, this data is recorded in .db files, one corresponding to each table in data model. These .db files are loaded into oracle database using some awk scripts during NETSMART installation. Once this data is loaded

into oracle database, during Application server startup this information is read by **Meta Model** engine and cached in memory in form of relationship tree. Once loaded into memory, **Meta Model** engine uses this cache for future refer-ence. It is possible to delay loading of this information into memory **Meta Model** engine is the executable part in **meta model** which implements all the interfaces exposed. It also implements semantic interpretation of **Meta Model** data, loading of **meta model** data from database to memory.

Detail Description Paragraph - DETX (1187):
[1264] In order to be able to support autodiscovery, **Meta Model** is dependent on support from COMMS for that NE type.

Detail Description Paragraph - DETX (1189):
[1266] This section addresses process of adding support for a new network element assembly. Following are main steps involved in supporting new network element in netsmart from **meta-model** point of view:

Detail Description Paragraph - DETX (1199):
[1276] This section defines the data model which is used to instantiate the **meta model**. The assemblies for all net-work element types supported by NETSMART will be stored in the database (Oracle). A specific NE **meta model** can be loaded into memory on demand or preloaded during the Application Server initialization. SQL queries are used to retrieve the assembly data and instantiate the **meta model**. The data model has been updated for the 2.i release. Changes for 2.i included (a) adding newer columns to existing tables (b) dropping columns which existed prior to 2.i and (c) introducing new tables. Table and column definitions which existed prior to 2.i are mentioned in italics. This is expected to serve as a reference for developers.

Detail Description Paragraph - DETX (1200):
[1277] The following sections describe the data model and how the **meta model** uses it to instantiate a network element assembly:

Detail Description Paragraph - DETX (1204):
[1281] The following tables are used to define all network element assemblies which are managed by NETSMART. The **meta model** is built by querying the following tables:

Detail Description Paragraph - DETX (1244):
[1321] The meta_property table captures the properties.sup.2 for a specific component. A property is a generic way of grouping attributes of a component. The grouping of attributes into properties is application .sup.2 Properties can be visualized as category of attributes. There are various categories of attributes modeled in **meta model** e.g. TL1 related attributes, attributes required to control behavior of other components etc. dependent. For example, the attributes which are used to describe a components current state and configuration are grouped into the ATTR_LIST property.

Detail Description Paragraph - DETX (1277):
[1354] This is a temporary table. It is used by **Meta Model** awhile populating its tables with data

Detail Description Paragraph - DETX (1285):
[1362] This section describes database queries performed by **Meta Model** to instantiate a network element assembly. The following sections describe how

the **meta model** is built:

Detail Description Paragraph - DETX (1331):
[1408] The result of this query are the successors to the specified parent and ne_base parameters. This SQL is recursively called in order to build the assembly hierarchy. It is started at the root MetaNe node and traverses depth first until it reaches leaf nodes and the entire tree is instantiated. NOTE: This is the most expensive, time consuming, part of building the **meta model**.

Detail Description Paragraph - DETX (1387):
[1464] **Meta Model**

Detail Description Paragraph - DETX (1388):
[1465] The MetaModelFactory is a singleton object which requires initialization in each process space that requires **meta model** access. The MetaModelFactory creates a MetaBaseAssembly for each base network element supported by the system (e.g. FACTR, FLM150, etc.). In addition, the MetaModelFactory assigns MetaAssembly(s) to each MetaBaseAssembly reflecting the network element revisons that are supported by NETSMART (e.g. FACTR 05 01, FACTR 05 02, FLM150 11 02, FLM150 12 02, etc.). Each MetaAssembly evaluates its auto load flag. If it is True, the MetaAssembly will initiate the instantiation of the assembly. The instantiation of the assembly starts at the MetaNe root node and instantiates a tree of MetaNode(s). The tree of MetaNode(s) include named relationships between nodes called MetaEdge(s).

Detail Description Paragraph - DETX (1395):
[1472] The MetaNode is an abstration which allows for generic/hierarchical traversal of a network element assembly. The MetaNe is the root MetaNode. The typed specific behavior is defined by the MetaObject subclasses. A MetaObject is a generalization of the things that comprise a network element. The MetaObject may contain properties, or lists of attributes, called MetaProperty(s). The MetaProperty contains MetaAttribute(s). The MetaProperty and MetaAttribute may contain a Condition. The Condition must be evaluated at run time with context. Only if the Condition is true does the MetaObject recognize that MetaProperty or MetaAttribute. A MetaObject is created by the MetaReflector using Java reflection. The MetaObject is a abstract **class** used to define generic **behavior** for typed network element **objects**. The network element assembly is a generic tree pattern implemented by the MetaNode and MetaEdge relationships. The network element assembly is also typed. Each MetaNode is implemented as a MetaObject and subclass. FIG. 41 shows the MetaNode class diagram.

Detail Description Paragraph - DETX (1398):
[1475] FIG. 42 depicts a small portion of a FACTR assembly. Starting at the root MetaNode (e.g. FACTR.sub.--5.sub.--2), the **meta model** is hierarchically built depth first. The root node contains a PHYSICAL_SIDE and a LOGICAL_SIDE. From the PHYSICAL_SIDE, the hierarchy continues by working its way down through the shelves, sections and equipments, etc.

Detail Description Paragraph - DETX (1399):
[1476] The hierarchy of the assembly is totally driven by the data model. The loading of the **meta model** is strictly a reflection of what has been modelled in the meta_assembly table.

Detail Description Paragraph - DETX (1401):
[1478] Objects & Relationships Modelled in **Meta Model**

Detail Description Paragraph - DETX (1402):
[1479] This section provides list of various objects and relashionships modelled in **meta model**. The list of objects and relashioships provided here covers most of the common scenarios which apply to most of network assemblies. It is important to understand that need for modelling new objects and relationships may arise in future depending on the assembly and the kind of support required in NETSMART for that assembly. Depending on the kind of support provided for an assembly, every relationships object may not be applicable. Following is the list of commonly modelled objects in **meta model**:

Detail Description Paragraph - DETX (1411):
[1488] In addition to that, there are some pseudo objects which can be modelled to organize the hierarchy better. These objects are Sections and Groups. Sections are primarily used to group related equipment together and Group is a more general notion to group set of related meta objects together in meta_assembly. Following is list of relationships modelled in **meta model**:

Detail Description Paragraph - DETX (1413):
[1490] CONTAINS_ONE_OF If a parent object in the containment hierarchy can have only one of possible children (in dynamic view) then this **relationship is used between parent object** and individual children objects.

Detail Description Paragraph - DETX (1438):
[1515] Relationship between two meta objects could be conditional. While **meta model** relationship tree traversal, relationships qualified with conditions are only traversed if the condition is met. Traversal logic does not reach a node, if there was a condition on its predecessor edge (i.e. relationship to parent) which is evaluated to `false`. Similar logic applies to traversal from a child to parent object as well. This kind of conditional relationships are covered in meta_assembly.

Detail Description Paragraph - DETX (1439):
[1516] Applicability of an attribute to a Meta Component or Meta Instance could be conditional. An attribute and all its characteristics (e.g. vaild values, read only etc) will apply to a Meta Component or Meta Instance only if the associated condition (if any) evaluates to true. The conditions in **meta model** look like DataObject.getAttributeValueString("CO- NF5")=="D1" This implies that if the value of attribute `CONF5` is equal to `D1` in an object of class DataObject then this will evaluate to `true`. In order to evaluate conditions, **meta model** needs `context`. A component which invokes an interface on **meta model,** needs to provide context as well so that **meta model** can evaluate any related conditions. `Context` is a generic concept and is a vector of objects. In the example of condition above, the `context` is expected to be a vector of DataObject.sup.4 objects. The framework component responsi-ble for evaluation of conditions, will invoke `getAttributeValueString("CONF5")` method on each object in con-text and compare the return value with `D1`. If there was any object in the context for which this comparison succeeds the condition will be evaluated to true. .sup.4 DataObject is a java class in NETSMART.

Detail Description Paragraph - DETX (1441):
[1518] This section presents overview of the autodiscovery process and involvement of **meta model** in this process. When user adds a network element and performs `login` from NETSMART GUI, ACT-USER TL1 command is sent by NETSMART to log into the network element. Autodiscovery process is triggered, once successful login and some basic initialization is achieved. Autodiscovery process can be divided into three major phases as described below:

Detail Description Paragraph - DETX (1442):
[1519] 1. Retrieving configuration information from the NE being autodiscovered--During this phase all con-figuration information is retrieved from network element by sending appropriate TL1 commands for retrieval (e.g. RTRV-EQPT etc.). Communication Server provides interfaces to the application server for these TL1 commands. As a result of invocation of these interfaces on COMMS, Application Server gets collection of dataobjects. COMMS returns one dataobject for each aid fetched from the network element. These dataobjects are created by COMMS after parsing the TL1 responses. Each KEYWORD-DOMAIN pair in TL1 response is translated to a Attribute-Value pair. Each dataobject contains collection of such Attribute-Value pairs. In addition to collection of Attribute-Value pairs, dataobject also contains aid and ComponentID.sup.5. During this phase information about all entities (i.e. equipment, facilities, EPGs etc) is retrieved from network element. This information retrieval is driven by DataCache. Their are subclasses of DataCache for each NE type supported by NETSMART. For example, there is FACTRDataCache which drives this information retrieval for FACTR NEs during Autodiscovery. It is responsibility of the **meta model** person who is modelling a new NE type to implement a subclass of DataCache for that NE type. From the dataobjects received from COMMS, the DataCache build a hashmap of these dataobjects with CompinentID being the key. DataCache also supports interface to obtain a dataobject based on its ComponentID, which is used by Autodiscovery in second phase. .sup.5 ComponentID is internal to NETSMART and can be treated as a unique identifier for a dataobject for a given TID. AddressTranslator in COMMS translates the aid in TL1 response to corresponding COmponentID by following some predefined rules.

Detail Description Paragraph - DETX (1443):
[1520] 2. Traversing the **meta model** relationship tree and creating application objects 2--Once the DataCache has retrieved all the information and built a hashmap of dataobjects, Autodiscovery traverses the **meta model** relationship tree and creates application objects. These application objects are passed to Configuration Manager, which holds and manages them. The traversal starts from the root node and for every node following steps are taken to create application objects:

Detail Description Paragraph - DETX (1450):
[1527] 3. Configuration Manager builds a containment tree of application objects created by Autodiscovery. The containment relations in this tree of application objects (also called as dynamic tree or dynamic view of the NE) are same as modelled in **meta model**. When autodiscovery passes an application object to configuration manager, it also passes reference of parent application object.

Detail Description Paragraph - DETX (1467):
[1544] The element manager applications include a **meta model** that describes an NE for a given release, autodiscovery that identifies and discovers NEs, configuration manager for the NEs, crossconnect manager that manages the NE crossconnects, a fault manager, and software download and remote backup manager for the NEs.

Detail Description Table CWU - DETL (5):
5 Table Column Datatype Description NE_BASE varchar2(32) The base network element identifier (e.g. FACTR). PARENT varchar2(32) A instance identifier for a specific assembly component. This instance of a component is the direct predecessor to the corresponding NODE identifier in the assembly hierarchy. NODE varchar2(32) A instance identifier for a specific assembly component. This NODE identifier is the successor to the PARENT identifier. EDGE varchar2(32) The EDGE defines a named relationship between the PARENT and the NODE. The reasons and usage of relationships is dependent upon

application usage. For instance the Autodiscovery application is primarily concerned with CONTAINMENT type relationships. Refer to appendix A for various relationships modeled in meta model. PRIORITY integer This determines the order in which rows are returned from database at the time of loading **meta model** into application memory. CONDITION varchar2(1024) The condition is an optional expression which can be attached to an edge. The dynamic evaluation of the conditional expression must be evaluated to true in order for the relationship between the PARENT and the NODE to hold.

Detail Description Table CWU - DETL (6):
6 Table Column Datatype Description META_CLASS_ID varchar2(32) An enumerated meta class identifier. META_CLASS varchar2(75) The actual JAVA class name, including package information. These classes are implemented in the **meta model** package and are instantiated using JAVA reflection. meta_assembly_map

Detail Description Table CWU - DETL (11):
11 Table Column Datatype Description NE_BASE varchar2(32) The base network element identifier (e.g. FACTR). META_ATTRIBUTE_ID varchar2(32) Unique identifier for the attribute. PROPERTY_ID varchar2(32) Prior to release 2.i, this column was called PROPERTY.sub.-- NAME.Together with NE_BASE, references meta_property(ne_base, meta_property_id) ATTR_NAME varchar2(32) The name assigned to this attribute. TYPE varchar2(10) The application type of the attribute. For example, String, Integer, Boolean, Float, Enum. VALUE varchar2(256) Although the data model, and **meta model** do not know the current state of any components attributes. Some attribute have a static value. This value is not expected to change. DEFLT varchar2(20) The default attribute value. VALID_VALUES varray called A list of valid values for this attribute. This is str_list_t currently limited to 15 elements of maximum length 15. MIN varchar2(20) A minimum value for this attribute. Used for Integer and Float only. MAX varchar2(20) A maximum value for this attribute. Used for Integer and Float only. INCR varchar2(20) A increment value for this attribute. Used for Integer and Float only. READ_ONLY varchar2(5) A boolean flag indicating whether this attribute may be changed or not. REQUIRED varchar2(5) A boolean flag indicating that this attribute is required.

Anthony Vetro, *Mitsubishi Electric Research Labs*
Charilaos Christopoulos, *Ericsson Research*, and
Touradj Ebrahimi, *Swiss Federal Institute of Technology—EPFL*

Universal Multimedia Access

Access to information, until recently, has only been possible through dedicated infrastructures and in a rather rigid way. Television and radio sets, connected to a limited number of broadcasters through terrestrial, cable, or satellite channels, allowed users to access programs and news mostly produced in a generic manner. The Internet and the World Wide Web (WWW) brought a new dimension to information access and a fundamental change: content democratization. Thanks to the Internet and the WWW, it has become possible to be both a producer and a consumer of content at the same time. Problems and challenges such as access to quality information, search and retrieval, as well as ownership rights have also appeared and constitute some of the challenges in future.

In parallel to these challenges, advances in signal processing combined with the appearance of heterogeneous networks are paving the way for users to enjoy services wherever they go and on a host of multimedia capable devices. Such devices include PCs, TVs, and mobile devices. Each may support a variety of formats and multimedia content. The networks they are connected to are often characterized by different conditions, and the terminals themselves vary in display capabilities, processing power, and memory capacity. Additionally, user preference, personalization, and other factors of the usage environment must also be accounted for. Given this complex and dynamic usage environment, it becomes necessary to consider methods of adapting the content accordingly. This framework, where information is accessed in a suitable form and modality, is referred to as universal multimedia access (UMA).

The UMA framework forces us to study the functionality of every component in the end-to-end delivery chain in the context of various use cases. This articles in this issue have been put together to present the state-of-the-art in multimedia adaptation and to provide an overview of the standards that will support the UMA framework. It also provides readers with a road map toward UMA applications and discusses its future trends.

First, it is important to understand the content representation format, i.e., how the content is compressed and coded. Scalable representation formats allow for simple scaling of the content, but it is also necessary to consider efficient methods for adapting nonscalable content formats. These topics are covered in the first two articles. For video, "Video Transcoding Architectures and Techniques: An Overview" is presented by Vetro et al., and audio is covered by Homayounfar in "Rate Adaptive Speech Coding for Universal Multimedia Access."

The description and management of content is another critical aspect within the UMA framework. An efficient way to describe multimedia content is provided by metadata standards such as MPEG-7, which support efficient search, filtering, and browsing applications, as well as means to represent content summaries and variations. "Metadata-Driven Multimedia Access," by van Beek et al., provides an overview of these relevant techniques, describing how they support the UMA framework.

Bormans et al., in "MPEG-21: The 21st Century Multimedia Framework," presents the vision of the MPEG-21 standard to enable transparent access to multimedia content. The various aspects of this standard are expected to play a key role in realizing the UMA framework. One important aspect of this effort is to develop a standardized description of usage environments to enable, for example, negotiation of device characteristics and QoS parameters. A key component of MPEG-21 to this respect is digital item adaptation, which is under development.

This issue concludes with an article by Pereira and Burnett that takes a step back to analyze existing trends in the related areas of communication and multimedia access and leads us through some future directions. In "Universal Multimedia Experiences for Tomorrow," the migration from today's universal multimedia access to tomorrow's universal multimedia experiences is presented.

In closing, the guest editors would like to thank all the contributing authors for their very timely contributions, the reviewers for their invaluable comments to ensure high-quality papers, and the editorial board for their engaging support. Please enjoy the articles in this issue!